Sparse Systems Solving on GPUs with GMRES

Raphaël Couturier, Stéphane Domas

J Supercomput
DOI 10.1007/s11227-011-0562-z

Sparse systems solving on GPUs with GMRES

Raphaël Couturier · Stéphane Domas

Abstract Scientific applications very often rely on solving one or more linear systems. When matrices are sparse, iterative methods are preferred to direct ones. Nevertheless, the value of nonzero elements and their distribution (i.e., the sketch of the matrix) greatly influence the efficiency of those methods (in terms of computation time, number of iterations, result precision) or simply prevent the convergence.

Among iterative methods, GMRES (Saad, Iterative methods for sparse linear systems. PWS Publishing, New York, 1996) is often chosen when dealing with general nonsymmetric matrices. Indeed its convergence is very fast and more stable than the biconjugate gradient. Furthermore, it is mainly based on mathematical operations (matrix-vector and dot products, norms, etc.) that can be heavily parallelized and is thus a good candidate to implement a solver for sparse systems on Graphics Processing Units (GPU).

This paper presents a GMRES method for such an architecture. It is based on the modified Gram–Schmidt approach and is very similar to that of Sparselib (Barrett et al., Templates for the solution of linear systems: building blocks for iterative methods, SIAM, Philadelphia, 1994). Our version uses restarting and a very basic preconditioning. For its implementation, we have based our code on CUBLAS (NVIDIA, http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUBLAS_Library_2.1.pdf, 2008) and SpMV (Bell and Garland, Efficient sparse matrix-vector multiplication on CUDA. NVIDIA technical report NVR-2008-004, 2008) libraries, in order to achieve a good performance whatever the matrix sizes and their sketch are. Our experiments exhibit encouraging results on the comparison between Central Processing Units (CPU) and GPU executions in double precision, obtaining a speedup ranging from 8 up to 23 for a large variety of problems.

Keywords GPU · CUDA · GMRES · Iterative methods

R. Couturier (✉) · S. Domas
LIFC, IUT Belfort-Montbéliard, University of Franche Comte, BP 527, 90016 Belfort CEDEX, France
e-mail: raphael.couturier@univ-fcomte.fr

Published online: 10 February 2011

1 Introduction

For the past 10 years, distributed computing has been taking an increasing part in research works, while "old fashioned" parallelism has more or less become limited to experimental results on engineering applications. The recent developments of multicore chips and specialized hardware, like GPUs, reverse this tendency, providing new research ways in high performance computing. As usual, new architectures mean new or adapted algorithms to achieve the best performances. And it is also true that algorithms must take into account data organization. It is well known that mathematical methods are different whether they apply on dense or on sparse matrices.

In dense linear algebra, one of the main problems to address is that of memory management. For example, executing a task in parallel on the two cores of an Intel Xeon may take longer than if only one core computes. This negative speedup mainly comes from the fact that a single memory bus feeds the cores with an insufficient bandwidth and chipset to support efficient interlaced accesses.

It is obvious that algorithms doing a lot of calculations compared to their volume of data should hide a low memory bandwidth better. But the main point is to have a fine granularity of calculations in order to use the local cache of each core efficiently. It is especially true on GPUs where a pool of threads may have access to a fast shared memory segment. Nevertheless, each thread must peek to particular locations of this segment in order to have a real parallel access. Otherwise, accesses are sequentialized.

In the sparse domain, we can intuitively say that this problem is magnified since the matrices are often stored in compressed schemes that imply memory indirections to access nonzero values. Thus, local caches are poorly used. Obviously, this effect can be minimized if the matrices are stored in dense blocks. Unfortunately, this solution is not advisable for a large variety of problems, for example, those leading to diagonal dominant or spray matrices.

In this case, a lot of libraries [1, 6, 10] propose sets of functions to solve sparse systems with various iterative methods (GMRES [13], Conjugate Gradient [13], ...). It is well known that each of these methods has advantages and drawbacks when compared to one another. A lot of efforts have been made to optimize the stability, precision, convergence speed, memory usage, etc., thus the overall performance of each method. Less efforts may have been made to improve the raw performance, that is the execution time, because a lot of "end-users" are more interested in having a reliable result than in optimizing the time consumption.

Obviously, computer science researchers are concerned and have, therefore, tried to parallelize these methods more or less successfully on different architectures. Libraries like PETSc [1] achieve really good performances on clusters, partly because the memory management problem discussed above is overwhelmed by the volume for computations and the need for global communications. Nevertheless, taking the same number of processors, the best performance was often achieved on shared memory machines, for which the memory management and the granularity of computations are the main problems to address to optimize the execution time. This is why it is particularly interesting to develop and test classical iterative methods on new architectures like GPUs.

The goal of this article is to show that GMRES can be implemented relatively easily on NVIDIA GPUs to obtain a $\times 20$ ratio out of a CPU execution. Section 2 briefly describes the NVIDIA GPUs architecture and its coding principles with CUDA [12]. Section 3 presents the GMRES algorithm and the requirements to adapt it for GPUs. Section 4 shows experimental results, comparing CPU and GPU executions for various matrices. Finally, Sect. 5 presents with more details some implementation choices we made and some other works related to system solving on GPUs.

2 GPUs architecture and coding

2.1 Architecture

Intrinsically, a GPU is composed of several multiprocessors, each of them being like a massive SIMD (single instruction, multiple data) machine. More exactly, each processor supports the concurrent execution of multiple threads, leading to a SIMT (single instruction, multiple threads) architecture.

In order to execute a *kernel* (i.e., a routine), the GPU uses a two-level data-parallelism approach. Basically, each thread executes the *kernel*, operating on different data. Threads are grouped by *blocks* and the GPU schedules the *blocks* over the multiprocessors, according to their available execution capacity. It implies that the programmer must take care to build *blocks* with no execution dependencies and that do not write to the same data.

When a *block* is given to a multiprocessor, it is split in *warps*, composed of 32 threads. *Warps* are scheduled with very fine granularity: at each instruction step, a warp is chosen and its next instruction executed. In the best case, all 32 threads have the same execution path and the instruction is executed concurrently. If not, the execution paths are executed sequentially, which greatly reduces the efficiency. As for *blocks*, *warps* execute independently readings and writings (by warps or threads of the same warp) to the same data lead to unpredictable values.

A GPU has several levels of memory. Each thread has a private local memory. Threads of the same *block* can share 16 KB of memory, organized in banks of 1 KB. Finally, all threads can access to the GPU global memory, which is often filled with data from the host memory, using special functions.

Data can be stored/retrieved from these memories in a few instructions cycles but the local and global memories are not cached and have a very big latency (several hundred cycles). Thus, copying data from the global to the shared memory before doing intensive computations can prove useful and efficient. Unfortunately, the shared memory is very small which implies having a lot of copies and thread synchronizations in order to process all the data. A good overlap between computations and global memory accesses may be more efficient. Nevertheless, whether global or shared, memory bandwidth mainly depends on the access scheme. In the best case, a *half-warp* (i.e., 16 threads) may access concurrently to memory and at worst, 16 accesses are needed.

It should be noticed that computing in double precision with sparse data, as in our tests, prevents us from obtaining the best efficiency. But even if we do not care about memory access optimizations, actual architectures can deliver a speed-up close to 23 compared to a CPU execution, on a GMRES, as shown in the experiments section.

```
__global__
void dot(int n, float *x, float *y, float *d) {

  int id = blockIdx.x * blockDim.x + threadIdx.x;

  if (id < n) %{
    *d += x[id] * y[id];
  }
}

void doADot()

  int n = 1024*1024;
  float *x, *y, *d;
  dim3 blockSize(512);
  dim3 gridSize(n / blockSize.x);

  // allocate and fill x,y,d on the device (GPU)
  dot<<<gridSize,blockSize>>>(n,x,y,d);
  }
```

Fig. 1 A "naive" (i.e., not working) dot product *kernel*

2.2 Coding

Coding for such architectures may appear a little bit tricky, especially when optimizations are aimed at. Basically, the developer must use the CUDA SDK, which contains, among others, libraries and a dedicated compiler `nvcc`. The code must be written in C, using special extensions defined by CUDA. For example, `__global__` is a function type qualifier that declares a function to be executed by the GPU device (i.e., a *kernel*) and called by the host. CUDA also proposes several functions and types optimized for the GPU, concerning memory management, arithmetic, and texture operations. Dealing with basic linear algebra operations can be achieved through the use of CUBLAS [11], a BLAS [9] portage for GPUs.

Each *kernel* must be called providing two special parameters: the *block* size and the grid size. Both can be in one, two, or three dimensions. The first one defines the number of threads within a *block*. The second one defines the number of *blocks* needed by the kernel to cover computation domain. Fixing their values highly depends on the application and the architecture limitations.

Basically, each kernel begins with a few lines to obtain a unique index for each thread, using "built-in" structures. This index is used to access the data needed by each thread in the following.

Figure 1 gives an example of a kernel that computes a dot product. As data are vectors, we only need one-dimension *blocks* and grid. The *block* size is chosen to the maximum value allowed by the architecture. The grid size, that is the number of *blocks*, is directly determined by the problem size divided by the *block* size. The first line of the kernel computes the unique index of each thread. This index is the same for `x` and `y` so that each thread computes a different part of the product. As each thread of a *half-warp* uses different but contiguous locations, global memory can be accessed concurrently by the 16 threads. This is not the case for `d` since each thread writes it: memory accesses will be serialized in an undefined order. Furthermore, as

accesses are non-atomic, some could fail, leading to a false result. In conclusion, a working dot product should NOT be implemented like this.

This example points out a very common pitfall and clearly shows that the difficulty to program a GPU resides in the decomposition of the problem into groups of very fine grain tasks, with as few dependencies as possible. Tasks within a group can be synchronized and/or use atomic functions, but it is impossible to synchronize tasks from different groups. Furthermore, each task must have the same execution path to obtain the best efficiency. It is very far from the message passing paradigm that is mostly used nowadays for high performance computing. It is closer to data parallelism but with special constraints.

Nevertheless, a few principles may greatly help the developer, if correctly applied:

- Coding is data-driven, that is, the developer does not care about the number of processors. He just cares about the *block* size, checking that it does not exceed the maximum authorized by its GPU unit. The *block* size will often represent a block of data of the same size to be computed.
- For maximal concurrency, each thread should compute different data.
- Each time a *kernel* does an operation of unknown duration, check if threads must be synchronized before processing on the following. For example, copying from global to shared memory implies a synchronization before using the shared data.
- Each thread within a *warp* should store results in a different location. If it is not possible, use atomic functions.
- To optimize memory bandwidth, each thread within a *warp* should access different but contiguous data.
- For maximal efficiency, shared memory should store data frequently used during the computation of a *block*.

Obviously, there could be many more advice, mostly directly imposed by the type of architecture used. Nevertheless, they would still be useless without a consequent expertise in parallel programming. This is why it is important to develop the counterpart of classical computing libraries for GPU architectures. And, in our opinion, a GMRES method for GPU is the perfect beginning to reach such a goal.

3 GMRES for GPUs

A lot of papers describe different versions of the GMRES method. We chose a very common one, based on the modified Gram–Schmidt approach, with *restarting* and a very basic preconditioning. Figure 2 shows the general algorithm which is similar to the one given in [3].

There are three important points to study in order to adapt this algorithm to GPUs. The first one is the preconditioning technique used in the algorithm in order to reach or accelerate convergence. The second one is to take care of the data dependencies and the associated constraints imposed by GPU computations. It means, for example, that some operations cannot be parallelized and must be achieved by a single thread. The last one is to detect the most time consuming operations, which must then be optimized for GPUs.

```
1. load A, b, M^{-1}, initialize x with zeroes, end ← false
2. r = M^{-1}b, β_0 = ||r||_2
3. while (!end) do
4.   v^1 = r/||r||_2, s = ||r||_2.e_1
5.   for i = 1, ..., m do
6.     w = M^{-1}Av^i
7.     for j = 1, ..., i do
8.       h_{j,i} = w^T.v^j
9.       w = w − h_{j,i}.v^j
10.    done
11.    h_{i+1,i} = ||w||_2
12.    v^{i+1} = 1/h_{i+1,i}.w
13.    for k = 1, ..., i − 1 do h_{.,i} = J_k.h_{.,i} done
14.    construct J_i using h_{i,i} and h_{i+1,i}
15.    s = J_i.s
16.    if (|s_{i+1}|/β_0 < ε) then
17.      call updateX(i)
18.      end ← true
19.    fi
20.  done
21.  if (!end) then
22.    call updateX(m)
23.    if (β/β_0 < ε) then end ← true
24.  fi
25. done

26. function updateX(k) do
27.   ∀i, j < k, H_{i,j} = h_{i,j}, and s̃_i = s_i
28.   solve y from Hy = s̃
29.   x = x + y_1.v^1 + y_2.v^2 + ... + y_i.v^k
30.   r = M^{-1}(b − Ax), β = ||r||_2
31. done
```

Fig. 2 A GMRES method with restarting and preconditioning

3.1 Preconditioning

Like most of the iterative algorithms dealing with sparse matrices, GMRES performance (and even convergence) depends on the matrices conditioning. Choosing or creating the best preconditioner for each matrix is a research problem in itself that is not addressed at all in this paper. Our goal is not to provide the best efficiency for particular cases but an average in most cases. By the way, we state that M^{-1} must be easy to compute in order to have only matrix-vector products and no need of $Ax = b$ solvers. For our experiments, we choose M as the A diagonal, which is straightforward to inverse and provides a relatively good preconditioning, in so far as the matrices are not too ill-conditioned.

3.2 Data dependencies

The key problem is to have an accurate knowledge of the basic operations used in the algorithm and their connections. It allows to detect the lines that require special care to be coded. For example, a basic idea is to develop a *kernel* for the whole `updateX()` function. Since each line needs the result of the preceding line, a thread synchronization must be done between each line. Furthermore, some operations can be very simply parallelized while some others are purely sequential. For example, line 27 takes the form of a copy of $\tilde{s}$ in y, that can be done in parallel. Line 28 is a back-solve process (using y and H) that is sequential and must be executed by a single thread. It is the same in line 13 for which the dependencies between the updated elements of $h_{.,i}$ totally prevent to parallelize the loop. This is not the case of line 29 that is also implemented with a loop, but for which each thread can compute an x_i in parallel.

3.3 Time optimizations

Profiling techniques can provide useful information about the time consumption of each part of an application. Nevertheless, in our case, such techniques are not necessary to find the most time consuming operations. Considering that n is the problem size, the restarting limit m, generally chosen in few tens, is very small compared to n. Thus, all operations implying H (of size $(m+1) \times m$) are negligible. Furthermore, storing the J_i matrices can be reduced putting the c_i and s_i coefficients in a $(m+1) \times 2$ array. It implies that operations on lines 13 to 15, 27 and 28 are not time consuming.

Looking at the loops, it is also easy to eliminate operations on lines 2, 4, 29, and 30, which are done, at most, once by restart. Obviously, 2 and 30 would be much more time consuming if M^{-1} could not be computed easily, implying the use of a solver. Nevertheless, they are nearly the same operation as in line 6 that is done m times by restart.

It leaves lines 6, 8, 9, 11, and 12. Obviously, sparse matrix-vector products (namely SpMV) on line 6 are time consuming operations even if A is very sparse. As shown in the fourth column of Table 3, it represents in average the half of all floating point operations during the solve. It is quite simple to develop a Compress Sparse Row (CSR) based SpMV for GPUs and this was our first approach. Nevertheless, several articles propose more efficient *kernels* using special storage. We chose to use the code from [5], and more especially their hybrid matrix storage (ELL plus COO), which gave better results for all our experiments on GPU. The reasons for this choice are given with more details in Sect. 5.

Lines 8, 9, and 11 are respectively a dot product, an *axpy,* and a norm, which are included and optimized in the CUBLAS library [2]. It should be noticed that the dot product and the norm are reductions. This type of operation is not so obvious to implement efficiently for GPUs since the result must be stored in a single variable. Operations on line 12 could be done with two calls to CUBLAS (copy and scaling) but it is simpler and more efficient to write a new *kernel* for it.

Apart from these lines, loading matrices (i.e., Harwell–Boeing format reading plus CSR and hybrid format transformations) is also time consuming. They must firstly

be loaded in the CPU memory and then copied in the GPU memory. This process is quite long, even if the matrices are relatively small because of the limited amount of memory on the GPUs. Nevertheless, this overhead cannot be included in performance evaluations for two main reasons. Firstly, solving a system is very often only a part of a complex scientific application. In this case, the solver uses, most of the time, an on-the-fly generated matrix, directly into memory. Secondly, even if the matrix is loaded from a mass storage, the performance is clearly driven by the hardware capabilities and by where and how the matrix is stored on this hardware. This is why the following experiments only show the execution time to solve the system and do not include the time to transfer/generate the matrix.

In conclusion, implementing a classical GMRES for GPU architectures could easily and greatly benefit from optimized *kernels* of well-known operations. It insures efficiency and a compact source code, close in size to the CPU version. Problems come from all the other operations that must be developed. For these, a slight misunderstanding on how the *kernels* are executed or a bad analysis of the data dependencies within an operation will ineluctably leads to false results in the worst case scenario or, at best, to a poor efficiency. This difficulty is another argument to quickly implement and release libraries for GPUs, proposing "high level" operations like solvers.

4 Experiments

In this section, we report the experiments we have performed to evaluate the speed of our implementation of the GMRES algorithm. In order to measure the benefit of using a GPU for solving such sparse linear systems instead of using traditional CPU, we have used a Xeon CPU X5482 3.20 GHz with 4 GB memory, hosting a NVIDIA GTX 280 with 1 GB memory. All results are for *double precision data*. On the one hand, it is a drawback to have an efficient access to memory in actual GPU architectures, but on the other hand, it is mandatory to reach a good precision, under $1e^{-7}$ in most cases. Indeed, the few tests we made with single precision data have shown that precision was limited to $1e^{-4}$ while the execution time was only 1.5 times faster. Thus, "doubles" are a better trade-off between performance and precision.

As usual in this type of tests, we took matrices issued from different scientific fields, with a large variety of properties. We chose quite huge matrices with a relatively high number of nonzero values. Associated with a medium restarting limit of 16, it leads to an average filling of the GPU memory. For example, solving the `cage14` problem implies nearly 645 MB of memory.

All those matrices are available on the Sparse Matrix Florida Collection [8]. Their properties are summarized in Table 1. For each matrix, the second column provides the problem size, that is the number of rows in A. As we only consider square matrices, the number of columns is equal to the number of rows. The number of nonzero elements is another important parameter that is given in the third column. The fourth column gives a brief description of the considered problem that provides this matrix. Finally, Fig. 3 gives the sketch of each matrix, illustrating where the nonzero elements are located. For some matrices, most elements, if not all of them, are close to the diagonal whereas for others, some elements are far from the diagonal.

Table 1 Description of the matrices used

Matrix	Nb. rows	Nb. nonzero	Description
af_0_k101	503,625	9,027,150	sheet metal forming
af_shell9	504,855	17,588,845	sheet metal forming
BenElechi1	245,874	13,150,496	No description
cage14	1,505,785	27,130,349	dna electrophoresis
ecology2	999,999	4,995,991	circuit theory applied
F1	343,791	26,837,113	stiffness matrix from an engine
FEM_3D_thermal2	147,900	3,489,300	fem 3D nonlinear thermal problem
G3_circuit	1,585,478	7,660,826	circuit simulation problem
Ga41As41H72	268,096	18,488,476	real-space pseudopotential method
hood	220,542	9,895,422	test matrix
pwtk	217,918	11,524,432	pressurized wind tunnel
thermal2	1,228,045	8,580,313	unstructured fem

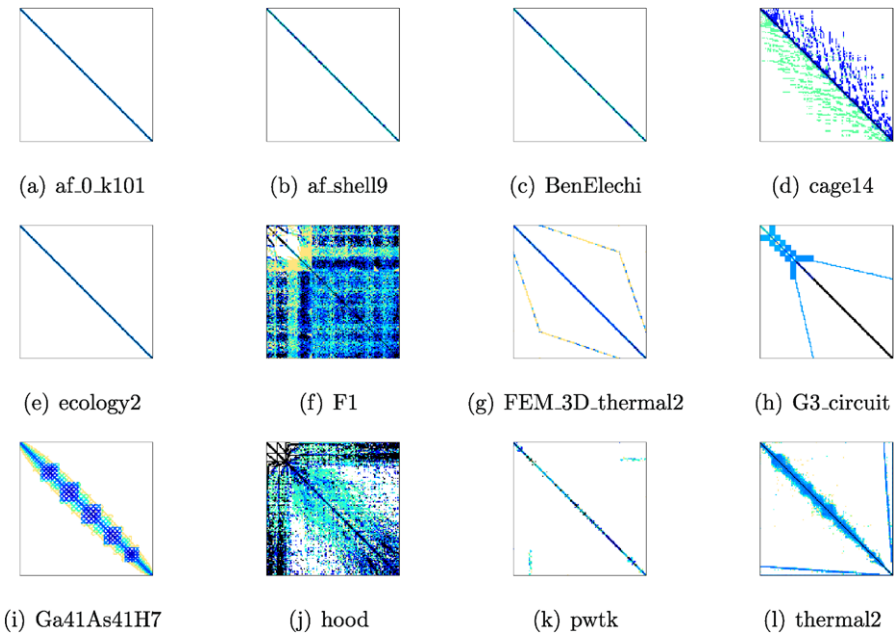

Fig. 3 Sketches of matrices used

In Table 3, we report the number of floating point operations per second (in Gflops) for each version of the solver (CPU and GPU). This number includes the total solving of the system. We also computed the percentage of flops used by the sparse product matrix vector during this solving. As shown in the fourth column, this percentage varies according to the structure of the matrix (size, number of nonzero elements, etc.). Compared to the results presented in [5], solving a linear system in GPU is

Table 2 Comparison between CPU and GPU, times in ms

Matrix	CPU times	GPU times	Nb. iter.	Ratio	Prec.	Δ
af_0_k101	4,076.34 ms	264.46 ms	49	15.41	1.86e–08	1.35e–20
af_shell9	5,238.20 ms	354.90 ms	65	14.76	2.34e–07	1.18e–20
BenElechi1	2,209.304 ms	202.63 ms	62	10.90	1.07e–08	4.16e–17
cage14	5,423.39 ms	310.51 ms	21	17.47	2.36e–09	2.45e–10
ecology2	4,100.96 ms	192.60 ms	35	21.29	1.59e–08	7.77e–16
F1	3,640.03 ms	415.94 ms	49	8.75	3.35e–08	2.71e–20
FEM_3D_thermal2	996.03 ms	112.62 ms	52	8.84	5.92e–08	1.36e–12
G3_circuit	6,370.25 ms	281.80 ms	33	22.60	6.96e–08	9.99e–15
Ga41As41H72	4,003.31 ms	500.49 ms	87	8.00	6.01e–09	2.58e–16
hood	1,721.11 ms	214.83 ms	56	8.01	1.58e–02	2.48e–17
pwtk	3,278.32 ms	327.76 ms	106	10.00	7.98e–04	3.55e–15
thermal2	3,031.47 ms	151.26 ms	20	20.04	3.77e–09	8.80e–14

Table 3 Comparison between CPU and GPU in Gflops

Matrix	CPU Gflops	GPU Gflops	% of SpMV in solving
af_0_k101	0.47	7.28	48
af_shell9	0.49	7.31	61
BenElechi1	0.66	7.28	61
cage14	0.42	7.43	51
ecology2	0.38	8.22	14
F1	0.57	5.05	67
FEM_3D_thermal2	0.68	6.11	56
G3_circuit	0.38	8.59	13
Ga41As41H72	0.66	5.30	65
hood	0.65	5.24	57
pwtk	0.69	6.87	59
thermal2	0.36	7.17	19

obviously less efficient than simply computing the SpMV. It can be easily explained by the fact that during the solving of a linear system, many operations are used in addition to the SpMV and some of them are not really efficient on GPU. For instance, all computations of dot products and norms require a parallel reduction which, by definition, cannot use all the threads of a GPU. Nevertheless, for our experiments, the number of Glops with the GPU varies between 5 and 8.

In order to validate our results we have always verified the solution of the system by computing

$$prec = \max(A \times X^{\text{cpu}} - b), \tag{1}$$

where X^{cpu} represents the solution vector computed by the CPU and where the function max finds the maximum among the components of a vector.

We also compute the difference between the CPU and the GPU solutions using

$$\Delta = \max |X^{\text{cpu}} - X^{\text{gpu}}| \tag{2}$$

where X^{gpu} represents the solution vector computed by the GPU.

In Table 2, we report the execution times in ms of both versions, for a threshold $\epsilon = 1e^{-10}$, a restart limit $m = 16$, and a right-hand side b filled with 1. We have always used the matrix format (and thus the SpMV *kernel*) that minimizes the execution time: hybrid format (ELL plus COO) for GPU and CSR format for CPU. The table also contains the number of iterations required to reach the threshold, the ratio between the elapsed time of CPU and GPU versions, the precision reached as expressed in (1) and the difference between both version as expressed in (2). Execution times represent only the solving time and do not include the cost of transferring and storing the matrices in the hybrid or CSR format.

These results clearly show that the GPU version, with the same level of precision as the CPU version is yet faster. It should be noticed that this precision is low for the `hood` and `pwtk` matrices, despite the small threshold ϵ chosen. Actually, these matrices are particularly ill-conditioned, which leads to the low precision and the slow convergence. Further tests should be done with more complex preconditioning matrices.

The number of iterations for both versions is always identical. The ratio of execution times between both version ranges from more than 8 up to 23. The larger the size of the matrix is, the closer the elements from the diagonal are, the faster the GPU version is compared to the CPU. We can also notice that performances are relatively independent from the structure of the matrices. In our examples, some are typical diagonal dominant whereas others are very spray. Likewise, matrices are issued from different scientific fields.

5 Related works

As shown in previous sections, the sparse matrix-vector multiplication is a critical operation to reach a good performance. Several works like [5], [4], and [15] propose efficient *kernels*, using special storages. Both [4] and [15] compare their results to the best *kernel* in [5] and it seems they obtain better performances in some cases. Nevertheless, their evaluation context is not clear and it is difficult to judge the pertinence of their results. For example, they never indicate if the computations are in single or double precision. [4] does not give the matrix sizes and the number of nonzero values. [15] gives it but the matrices that were chosen are generally small compared to the memory available on the tested GPU. Furthermore, they do not talk about what is measured, notably if they include or not the cost of the matrix transfer into memory and possible data reorganizations to fit a particular format. [5] is far more detailed about these points and achieve a very good performance when calling their *kernel* based on an hybrid storage (ELL plus COO, both these formats are described in [5]). This is why we have used this routine for the experiments conducted in this paper.

Other studies have been completed to design efficient sparse linear solvers for GPU. In [7], authors present Concurrent Number Cruncher (CNC), a general symmetric sparse system solver, based on the conjugate gradient method. The storage of

sparse elements is based on blocks. In many case, the use of hybrid storage is more efficient. Moreover, the conjugate gradient method is less general than the GMRES one.

In [16], authors present another GMRES implementation on GPU. Their experiments are done on matrices that are not available. Moreover, they do not give information on the structure and on the number of nonzero elements of the matrices they used. They do not obtain the same number of iterations with their CPU and GPU implementations while their precision is relatively poor.

In [14], authors describe the design and the implementation of the Jacobi and the bi-conjugate gradient method on GPU. Those methods are less efficient and general than the GMRES method and they only consider diagonal matrices.

6 Conclusion and perspectives

In this paper, we have presented an efficient implementation of GMRES for GPU. We have reminded the principles of GPU architectures and their coding in order to point out the main difficulties of such an implementation. Apart from the experimental results, the coding process has comforted us in the fact that the expertise level required to develop applications for GPUs is relatively high. This is the main motivation to release libraries of *kernels*, containing high level operations like solvers.

Concerning the experimental tests, they are in the direct line of other results concluding that GPUs completely overwhelm the CPUs in the linear algebra domain. In our case, we have obtained a maximum ratio between CPU and GPU executions close to 23 and an average of 13.8 over all our experiments.

Obviously, our work has some limits. The first one is the very basic preconditioning we chose. The next step is to use more complex M matrices and a solver to compute w and r (lines 2, 6, and 30). It is not clear yet if much better execution times would be obtained in the general case. Nevertheless, we could use ill-conditioned matrices and insure at least the convergence and a good precision.

We also plan to develop a parallel version of the GMRES algorithm using a cluster of GPUs. This would allow scientists to solve efficiently larger systems. Moreover, we plan to use our solver with scientific applications which require to solve many sparse linear systems.

Acknowledgement This work has been supported by the council of the Franche Comte region.

References

1. Balay S, Buschelman K, Gropp WD, Kaushik D, Knepley MG, Curfman McInnes L, Smith BF, Zhang H (2001) PETSc Web page. http://www.mcs.anl.gov/petsc
2. Barrachina S, Castillo M, Igual FD, Mayo R, Quintana-Orti ES (2008) Evaluation and tuning of the level 3 CUBLAS for graphics processors. In: IPDPS. IEEE, New York, pp 1–8
3. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, van der Vorst H (1994) Templates for the solution of linear systems: building blocks for iterative methods, 2nd edn. SIAM, Philadelphia
4. Baskaran MM, Bordawekar R (2009) Optimizing sparse matrix-vector multiplication on GPUs. IBM research report RC24704, IBM, April 2009

5. Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. NVIDIA technical report NVR-2008-004, NVIDIA Corporation, December 2008
6. Blatt M, Bastian P (2006) The iterative solver template library. In: PARA, vol 4699. Springer, Berlin, pp 666–675
7. Buatois L, Caumon G, Lévy B (2007) Concurrent number cruncher: an efficient sparse linear solver on the gpu. In: High performance computation conference (HPCC). Springer lecture notes in computer sciences, vol 4782. Award: Second best student paper
8. Davis T (1997) University of Florida sparse matrix collection. NA Digest, see http://www.cise.ufl.edu/research/sparse/matrices/
9. Dongarra J, Duff IS, Sorenson DC, van der Vorst H (1998) Numerical linear algebra for high-performance computers. In: Software, environments, and tools, vol 7. SIAM, Philadelphia
10. Dongarra J, Lumsdaine A, Niu X, Pozo R, Remington K (1994) A sparse matrix library in C++ for high performance architectures. In: Second object oriented numerics conference, pp 214–218
11. NVIDIA (2008) NVIDIA CUBLAS library 2.1. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUBLAS_Library_2.1.pdf
12. NVIDIA (2009) NVIDIA a programming guide 2.2. http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf
13. Saad Y (1996) Iterative methods for sparse linear systems. PWS Publishing, New York
14. Jost T, Contassot-Vivier S, Vialle S (2009) An efficient multi-algorithms sparse linear solver for GPUs. In: International conference on parallel computing, ParCo2009, September 2009
15. Vazquez F, Garzaon EM, Martinez JA, Fernandez JJ (2009) The sparse matrix-vector product on GPUs. Research report, University of Almeria, June 2009
16. Wang M, Klie H, Parashar M, Sudan H (2009) Solving sparse linear systems on NVIDIA Tesla GPUs. In: 9th international conference computational science—ICCS. Lecture notes in computer science, vol 5544. Springer, Berlin, pp 864–873